CSE 331 – Section 4 – Test Design Sample Solution

Suppose we have a class that stores a sorted list of Strings, possibly containing duplicates. When a new String is added to the list, it is inserted in the proper place so that the strings in the list are maintained in non-decreasing order. Here is the interface:

```
public class SortedStringList {
  public SortedStringList() {...}

  public boolean isEmpty() {...}
  public int size() {...}
  public int indexOf(String elt) {...} // return -1 if not found
  public String get(int pos) {...}     // first position is 0,
      // throw IndexOutOfBoundsException if pos<0 or pos>=size()

  public void add(String elt) {...}
}
```

Your job is to create a test suite for the `SortedStringList`. Your tests should only use the operations given above. Start with the most basic tests you can think of, and, with each successive test, marginally increase the complexity of the test over the preceding tests. For each test, you only need to describe the test setup (inputs) and expected result(s) (outputs). You should not write JUnit code – just give the basic logic of the tests.

Remember: each test should, as much as possible, only test for one new thing, so that if a test fails, the failure provides specific clues for where the error might be located. A first example is given below to help get started.

Input: create new empty list s
Expected: s.size() = 0

Additional tests:

// empty lists
Input: create new empty list s
Expected: s.isEmpty() = true

Input: create new empty list s
Expected: s.indexOf("abc") = -1

Input: create new empty list s; execute s.get(0)
Expected: IndexOutOfBoundsException thrown

(continued on next page)

Below, multiple tests are written together to save space. When writing actual JUnit tests, it sometimes makes sense to check several results in a single test method. But it's often a better idea to have each individual test check for one or a few specific results and then group related tests in a single class, using methods labeled with @Before/@BeforeClass and @After/@AfterClass annotations to perform setup operations that are the same for each test. Make sure to keep in mind granularity of your tests.

Lists with one element:
Input: create list s; s.add("abc")
Expected: s.size() = 1
s.isEmpty() = false
s.indexOf("abc") = 0
s.indexOf("xyz") = -1
s.get(0) = "abc"
s.get(1) throws IndexOutOfBoundsException
s.get(-1) throws IndexOutOfBoundsException

Lists with multiple elements added in order:
Input: create list s; s.add("abc"), s.add("pqr");
Expected: s.size() = 2
s.isEmpty() = false
s.indexOf("abc") = 0
s.indexOf("pqr") = 1
s.indexOf("xyz") = -1 // after last element
s.indexOf("aaa") = -1 // before first element
s.indexOf("def") = -1 // between
s.get(0) = "abc"
s.get(1) = "pqr"
s.get(2) throws IndexOutOfBoundsException
s.get(-2) throws IndexOutOfBoundsException

List with multiple elements added out of order:
Input: create list s; s.add("pqr"), s.add("xyz"), s.add("abc")
Expected: s.size() = 3
s.isEmpty() = false
s.indexOf("aaa") = -1
s.indexOf("abc") = 0
s.indexOf("def") = -1
s.indexOf("pqr") = 1
s.indexOf("tuv") = -1
s.indexOf("xyz") = 2
s.indexOf("zzz") = -1
s.get(0) = "abc"
s.get(1) = "pqr"
s.get(2) = "xyz"